Christopher Schmidt • Christine Koppelt

Kubernetes Resource Management

How it works • What to do

FAQ & RECIPES

INNOQ

Resource Management in Kubernetes

How it works - What to do

Christopher Schmidt
Christine Koppelt

ISBN —
innoQ Schweiz GmbH
Gewerbestr. 11 · 6330 Cham · Switzerland
Phone (+41) 41 743 01 11 · WWW.INNOQ.COM

Layout: Tammo van Lessen with XeLaTeX
Design: Sonja Scheungrab
Print: Pinsker Druck und Medien GmbH, Mainburg, Germany

Resource Management in Kubernetes – How it works - What to do
Published by innoQ Schweiz GmbH
1st Edition · November 2021

Contents

1 Intro

Kubernetes (K8s) is an open-source platform for managing containerized applications and services. Over the last couple of years, it has arguably become the most popular infrastructure platform with a rapidly growing ecosystem.

Its strengths are:

- It is supported by nearly every cloud provider and runs on-premise as well, which makes it possible to easily port software between different types of environments.
- It provides a uniform operating platform across language and framework boundaries.
- It provides sophisticated mechanisms to scale applications up and down automatically, vertically as well as horizontally, which enables efficient use of server nodes.
- It maximizes resource utilization by automatically relocating applications to other nodes if free resources are available there.

In this primer, we provide an overview of the parameters that can be adjusted to ensure that Kubernetes can run your workloads as efficiently as possible.

1.1 How Does Kubernetes Work?

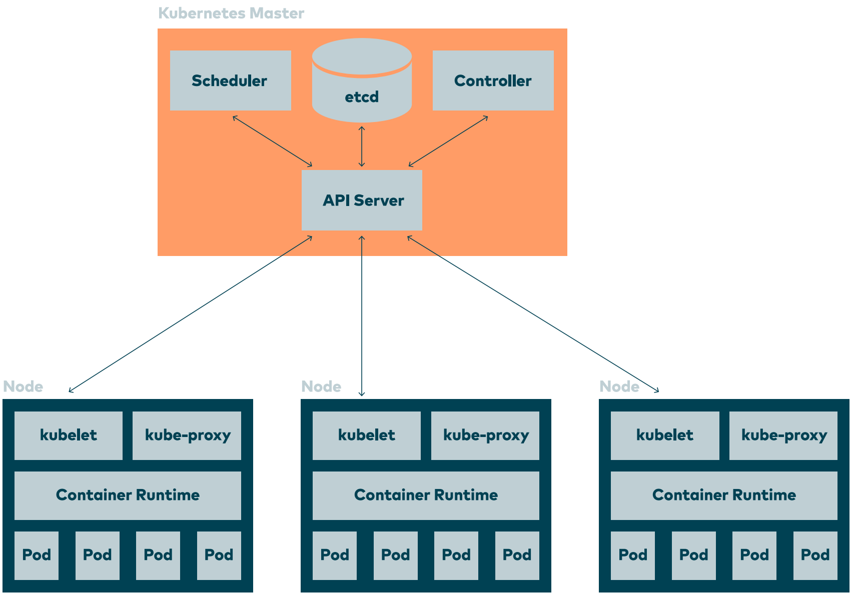
But before we dive into CPU and memory tweaking, let's have a look at the components of a Kubernetes cluster first. The brain of a Kubernetes cluster is one or multiple master nodes which manage the cluster. Besides this, it has a couple of worker nodes that run the workloads. Each of them has one or multiple Pods. A Pod is an entity with a unique IP address during its lifetime, which encapsulates one or more application containers, and, if needed, shared storage volumes.

Each of the master nodes comes with three parts:

- **Scheduler**: Selects a worker node for newly created Pods which are not assigned to a node yet.

- **API Server**: Exposes the Kubernetes API which is used for all automatic and manual management tasks.
- **Controller Manager**: Manages several administrative processes. The most important ones are:
  - Node controller: Removes worker nodes from the cluster and monitors the status of the existing nodes.
  - Job controller: Manages the Pods executing finite tasks (jobs).
  - Endpoints controller: Connects Pods to services. A service represents a group of Pods and exists longer than the Pods themselves, which may be restarted or redeployed. It provides a stable IP address and load balancing.
  - Service account and token controllers: Assigns API tokens and default accounts to new cluster namespaces (virtual clusters within a physical cluster).

Additionally, each master node typically runs an etcd instance for the cluster data.



Each worker node includes the following components:

- **Kubelet**: Communicates with the API server and manages the Pods on the node (including volume mounts, health checks, secrets, etc.).
- **Kube-Proxy**: Enforces rules for network communication of Pods from inside and outside of the cluster.
- **Container Runtime**: Each worker node requires a container runtime for the application containers. Kubernetes supports several: Containerd, CRI-O, and any implementation of the Kubernetes CRI (Container Runtime Interface) – Docker is deprecated with Kubernetes version 1.20.

1.2 The Challenge

Once we deploy our applications they consume resources like CPU, memory, network, or disk. The promise of Kubernetes is that the mapping of application resource demand to real existing hardware respective virtual machines is much easier. Yes, it is easier, but still a challenge. Due to the complexity of the applications, their programming languages and frameworks, their deployment and demand, etc., a wide range of controls have been created to choose from. Kubernetes supports means like Pod Disruption Budget, Taints and Tolerations, Resource Limits, Resource Requests, a Horizontal Pod Autoscaler, and many more. Linux itself adds the Completely Fair Scheduler (CFS), Control Groups, and Namespaces.

Our main goal in using Kubernetes is automation without interaction with an administrator. All these means, configuration parameters, and settings define how applications can run in an environment that is constantly changing, while the application itself is constantly changing.

2 Frequently Asked Questions

Frequently asked questions about how things work.

2.1 What Is a Completely Fair Scheduler?

A Completely Fair Scheduler (CFS) is a proportional share process scheduler that proportionally divides CPU time (CPU bandwidth) on a Linux box between groups of tasks (containers and/or Pods) according to the priority/weight of the task or the shares assigned to the Pods. Its goal is to maximize overall CPU utilization while maximizing interactive performance. Normally it is not possible that a user process blocks all CPU resources exclusively. Instead, the CPU resource distribution is weighted according to current demand.

2.2 What Is a cgroup?

Control groups are a Linux feature that allow processes to be organized into hierarchical groups whose usage of CPU and memory can be controlled.

Kubernetes for example creates a dedicated cgroup **kubepods**,

- containing cgroups that are defined by the quality of service (QoS) classes
- containing cgroups for the respective Pods
- containing cgroups of the Pod's containers

However, due to the hierarchical nature of cgroups, the cgroup **kubepods** is not the only deciding factor in how CPU resources are allocated to containers and the rest of the system.

2.3 Who Uses the Pod's Resource Requests Setting?

Above all the Kubernetes scheduler. It searches available nodes based on the given Pod resource CPU and memory requests. If it finds one, the Pod is scheduled to that node. If not, it checks if it can evict Pods based on their quality of service class or their Pod priority and preemption to gain more free space. If it is successful, the Pod is scheduled and started; if not it stays in state pending.

Also, this kind of pending state is a sign that the cluster ran out of resources. It can be used to initiate the addition of more worker nodes to the cluster.

2.4 Do Resource Requests Only Affect the Kubernetes Scheduler?

Actually no. Once the Pod is started, the CPU resource requests are also used to set the CPU share parameter of the CFS (defined by cgroups) for every container running in that Pod. CPU shares contain an integer value that specifies a relative share of CPU time available to the containers in a Pod. So if you set the CPU resource request of container A to 200m and of container B to 100m, A will receive twice the CPU time for tasks as container B (full container load).

However, since the absolute amount of CPU is dependent on the overall demand of all Pods and processes on the node, the actual amount of CPU is difficult to predict. Resource requests should therefore not be confused with a limit; they are rather a weighting. So container A can get even more CPU than 200m if CPU is available and requested by the processes running in A. On the other hand, if there is another Pod on the same node with one container and a request of 500m, the CPU is weighted between the two Pods by 3 to 5 (100m + 200m versus 500m) under full container load.

If you do not set any Resource Request in your Pod YAML, the CPU share is set to a very low value (at the time of writing it is hardcoded to 2), and your container will get almost no CPU (again at full container load. However, this can be a valid setting for batch processing Pods which only get CPU when other application Pods do not need it).

2.5 Is There a General Difference between CPU and Memory?

Besides the fact that these are different kinds of resources? Yes. Memory is a so-called incompressible resource. This means that every running process always sees the entire memory of the node, no matter what limits you set. However, it can only use the limited amount, because if it consumes more, it will simply be killed by Linux (OOM Killer).

CPU however is a compressible resource. So processes can be throttled. Of course, you can only use a CPU completely or not at all. For this reason, the CPU is distributed over a share of so-called CPU cycles. If you set the CPU limit to 50% of a core, the process will only be allocated 50% of the CPU cycles of one core.

2.6 How Do We Limit Container Memory?

Use memory limits, which are then propagated to a cgroup parameter of the container. Every process sees, however, the entire node memory. To effectively avoid OOM killing, we must rely on the framework that is used to finally limit its memory usage.

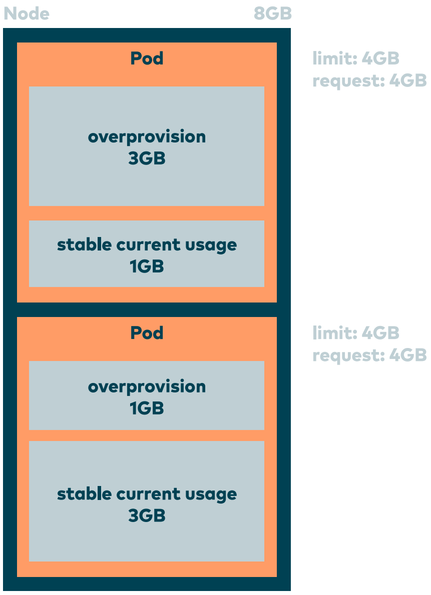
For instance the Java base image *openjdk:8u191-jdk* has built-in docker support. The JVM reads out the value of its cgroup memory limit parameter and sets the MaxHeapSize to 25% of the given memory limit, which is quite conservative. This is of course only the default and can be customized, with *XX:MaxRAMPercentage* for example. However, this limits only the Java heap memory. Whenever we create an object, it is always created in heap memory and under normal circumstances this is the largest memory. But it is not the only one. Shares between them depend on the type of application. To avoid OOM killing we have to include a memory spare and set the *MaxRAMPercentage* to ~80% for example. This leads to a, for sure required, over-provisioning of ~20% of the memory defined by the Java framework itself.

2.7 How Do We Limit Container CPU?

The CPU is a compressible resource, so it can actually be throttled. We can use CPU limits for this. Of course, a process can have the CPU or not. That's why CPU limits are implemented by CPU bandwidth control with *cpu.cfs_period_us* and *cpu.cfs_quota_us* parameters. These properties define a period, which is usually 1/10 of a second, or 100,000 microseconds, and a quota which represents the maximum number of slices in that period that a container process is allowed to run on the CPU.

Depending on the CPU cycles required to calculate for example a response by a service, some calls fit into that period, some do not. That is the reason why we see that not all service latencies are increased but only some (e.g. in the 90% percentile). However, latencies will increase while we reduce CPU limits. Also, some containers have a CPU-intensive initialization phase at startup. CPU limits also cover this phase, so that the startup times until the container is ready for operation increase.

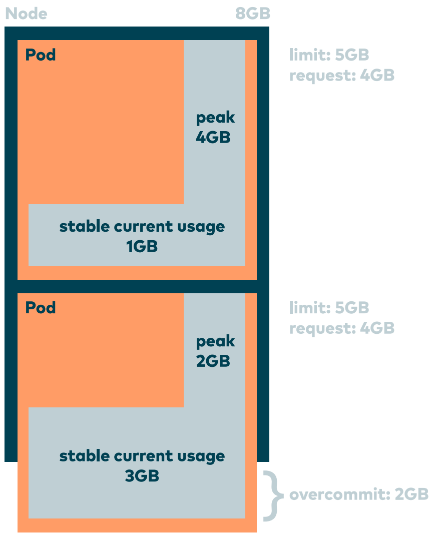
2.8 What Is Overprovisioning?



Overprovisioning is used to provide more CPU, memory, disk, or network resources than necessary, typically because clusters have to be big enough to handle peak demand. A certain safety buffer should be included, because you never know exactly how an application will respond to a load that you cannot predict. This is usually the case for web traffic. It also depends on the requirement for MTTR[1] to respond fast because of node failures, for example.

[1] mean time to recover, e.g. after a Pod crashed

2.9 What Is Overcommitment?



The Kubernetes scheduler ensures that the sum of all resource requests is lower than the node capacity. If we specify resource limits that are higher than resource requests, then the sum of the limits minus the node capacity is the amount of the overall overcommitment.

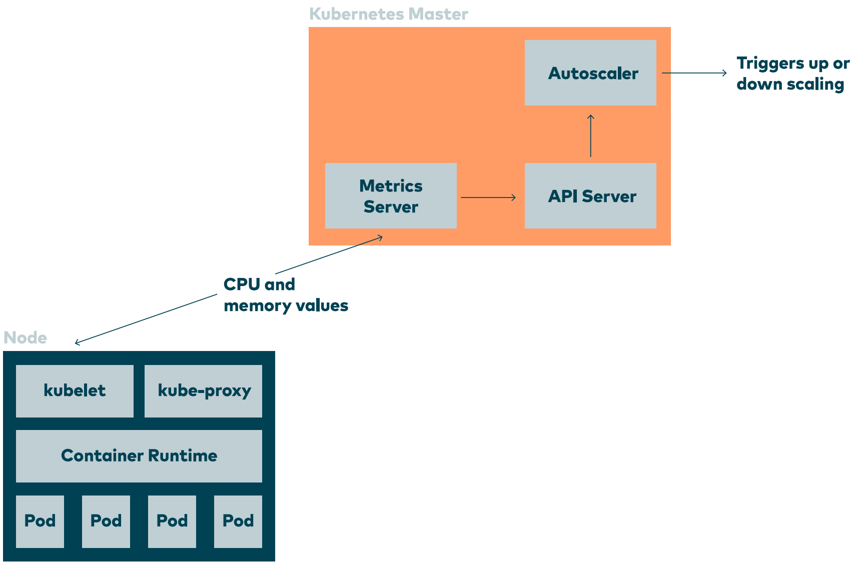
The idea of overcommitment may seem dangerous because containers will crash if, for example, memory is used up. In actual practice, however, overcommitment is a kind of compromise for workloads that need a much higher limit while initializing than while operating. In addition, allowing overcommitment appears to increase the average resource utilization of the nodes.

2.10 How Is High Load Noticed?

High load and the need for autoscaling are detected via resource metrics. The simplest way to provide such metrics is via the Kubernetes own metrics server – a lightweight, in-memory, Kubernetes native component, built exclusively for autoscaling purposes. Many cloud providers include it in their default setup. It queries each node's Kubelet instance for CPU and memory metrics of each Pod, aggregates them, and provides them to other components via the metrics.k8s.io API. It's also possible to provide additional metrics via a monitoring solution like Prometheus.

2.11 How Is High Load Responded To?

First, applications use their thread pools or other framework-related means to respond to higher loads. In any case, the processor load of the application will increase. With this increase, latencies and response times will increase. This behavior is of course not linear to the processor load, but the CPU is a good indicator.



CPU and memory are monitored via the metrics server. They can be used to trigger up- or downscaling via a specialized autoscaler, depending on your configuration. Basically, there are three options:

- Add more Pods for the application (Horizontal Pod Autoscaler).
- Increase the resource demand of the application Pod(s) (Vertical Pod Autoscaler).

- Add more nodes to the cluster to make more room for Pods (Cluster Autoscaler).

2.12 What Is a Horizontal Pod Autoscaler?

This is a fast response to higher CPU load by the Horizontal Pod Autoscaler (HPA) that creates instances of Pods. The HPA automatically scales the number of Pods that are defined by a deployment or StatefulSet, for example. Once the autoscaler adds additional Pods, the number of available resource requests and limits of the application to which the Pod belongs also increase. The HPA is thus a kind of autoscaler for application-based resource utilization.

2.13 What Is a Vertical Pod Autoscaler?

A Vertical Pod Autoscaler (VPA) frees users from the necessity of setting up-to-date resource limits and requests for the containers in their Pods. It will set the requests automatically based on usage and thus allow proper scheduling onto nodes so that the appropriate resource amount is available for each Pod. It will also maintain ratios between limits and requests that were specified in the initial container configuration.

2.14 What Is a Cluster Autoscaler?

A Cluster Autoscaler is a tool that automatically adjusts the size (amount of worker nodes) of the Kubernetes cluster. Current implementations react to Pods that are unable to be deployed (pending) due to resource restrictions. So on a busy cluster, a scaling request by an HPA may result in a scaling request by a Cluster Autoscaler, which typically takes minutes instead of seconds.

2.15 What Is a Cluster Proportional Autoscaler?

A Cluster Proportional Autoscaler watches over the number of schedulable nodes and cores of the cluster and resizes the number of replicas for a specific resource. This may be desirable for applications that need to be autoscaled with the size of the cluster.

2.16 Are Containers Secure?

Typical container runtimes start the processes of a container as normal Linux processes. They are isolated from other processes but not virtualized. Container processes share the kernel, I/O, network, and memory. One result of this is that a compromised container can also threaten other containers or the host itself. So the answer is no if you compare it to per application-process virtualization. However, since Linux processes have to be secure anyway, you can also say yes if you take care to restrict process capabilities and application and base-image vulnerabilities.

2.17 What Is a RuntimeClass?

RuntimeClass is a feature for selecting the container runtime configuration. The container runtime configuration is used to run a Pod's containers. Different RuntimeClasses for different Pods are possible to provide a balance of performance versus security. For example, if part of your workload comes from a third-party vendor (which you consider an untrusted component), you can run it with a container runtime that uses hardware virtualization (e.g. Kata Container, resulting in a small performance and resource penalty). Your own workloads are run by a standard runtime (e.g. Containerd, which is fast and with little overhead). Or something in between, for example a runtime (gVisor) that tries not to be a VM but still almost achieves its security by implementing a system API based directly on host system API primitives.

2.18 What Is a Probe?

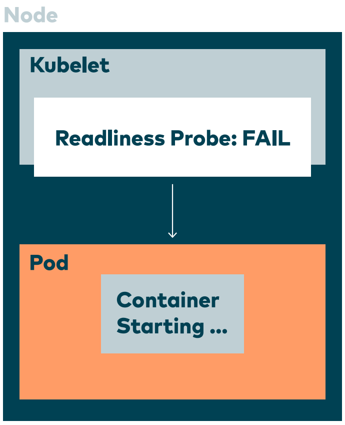
A probe is a manually implemented container endpoint (Socket, HTTP, or binary execution) that allows the Kubelet to check the internal state of the container.

2.19 What Is a Pod Disruption Budget?

A Pod Disruption Budget limits the number of Pods of a replicated application that are down simultaneously from voluntary disruptions. Typically there are no automated voluntary disruptions, but they occur through manual intervention. However, Pod Disruption Budgets cannot prevent involuntary disruptions from occurring (node crashing, VM issues, etc.). Writing disruption-tolerant workloads is essential anyway because of the up- and downscaling of clusters and Pod replicas.

3 Kubernetes Probes

A Pod is the smallest unit in Kubernetes. It is a container for containers that are running in a shared context like the same host, same IP, etc. The status of the containers can be checked by so-called probes. The respective results are then aggregated to the status of a Pod by Kubernetes. A probe is a diagnosis that is made regularly by the Kubelet on a running container. To perform this diagnosis, the Kubelet calls an endpoint implemented by the container process or executes a binary in the container. The Kubelet can perform and react to three types of probes: readiness, liveness, and startup.



Typical questions around probes are:

- Do we need separate endpoints for liveness and readiness probes?
- Do we always need all of them?
- Should the probes code-check the availability of upstream services?
- How are exceptions handled?

But first we need to understand how the lifecycle of a Pod is defined.

3.1 Phase and State of a Pod

A Pod's lifecycle is divided into two parts: a Pod phase, which is a simple, high-level summary of where the Pod is in its lifecycle, and the Pod state, which is an array of conditions through which the Pod has or has not passed. In addition there is a container state. A state is quite simple and can be waiting, running, or terminated. The Pod phase can be viewed by issuing `kubectl get pods` for example. The detailed Pod state can be seen by means of `kubectl describe pod <Pod name>`.

But what does this mean for the readiness of a Pod? If a Pod is in phase running, it means that at least one container is in the state running. But in the case of a multi-container Pod, it is not sufficient to reflect the state of only one container. Actually, the Pod condition is only ready once all containers are in state running. If this is the case, the Pod can be added to the load-balancing pool of all matching services. Otherwise, it is removed.

This procedure ensures that most cases of container creation or deletion are automatically handled correctly. The only thing we have to take care of is the time between container state running (that Kubernetes can use directly) and the ability to actually serve requests by the application code. This initialization time can be reflected by the readiness probe. In these simple cases it can use the same endpoint as the liveness probe.

3.2 Readiness Probe

Periodic probe of container service readiness. Containers will be removed from the service load balancer if the probe fails.

Recommendation: Use the readiness probe...

- ... during the container startup phase
- ... if an application takes itself down for maintenance

Question: Should a readiness probe check the application dependency?

3.2.1 Scenario

Besides its own ability to answer requests, the readiness probe of three replicas of an application is checking access to an upstream database service. In case of a database unavailability, all application replicas are then removed from load balancing since their dependency fails. In effect, the application is then offline (with no difference to “all Pods are failing” or to “deployment is deleted”). Imagine that all services in a system behave in the same way. The overall result is a propagation of faults that eventually become a system failure. The attempt to avoid this propagation is the main reason for not checking dependencies. Remember also that readiness in the sense of Kubernetes means: technically ready, not business ready. The probe only signals whether or not the Pod is added to the load balancer. A removed Pod always means failure, and this can never be a valid business status.

What should happen if the database is not available? As mentioned above, to avoid fault propagation, it is not advisable to simply put the readiness probe to false. One option is to implement some sort of degraded mode. For instance, a REST service answers only some requests that can be answered from cache or with a default, while responding with a 503 (Service Unavailable) on writes (PUT/POST). For sure we have to take care that downstream services are aware of this kind of degraded mode (in general, the downstream services should in any case be resilient to faulty calls to upstream services).

For the sake of completeness: A disadvantage of the degraded modes may be that they tend to end up with a kind of distributed degraded mode that is sometimes difficult to handle. So replying with 503 for everything may be a good option too.

How are exceptions handled? If the application code encounters an unexpected and unrecoverable internal exception while calculating the readiness response, it should crash on its own. This is because it can be expected to be a serious container-internal issue that has no connection with external dependencies.

3.3 Liveness Probe

Periodic probe of container liveness. Container will be restarted if the probe fails.

The liveness probe should be used if:

- The process in your container is unable to crash on its own whenever it encounters an issue or becomes unhealthy.
- Application code is running a framework where it is unable to control its execution (e.g. servlet container).

Concerning the verification of upstream dependencies, the same applies as for the readiness probe. Liveness probes should only help to determine whether the container process is responding or not. If the container process is able to detect its unhealthiness on its own, it can simply exit.

3.4 Startup Probe

Indicates that the Pod has successfully initialized. If specified, no other probes are executed until this completes successfully. Similar to the liveness probe the Pod will be restarted if it fails.

This (alpha feature) probe has been introduced to reflect long boot times typically experienced by legacy applications or technologies with uncomfortably long initialization times such as Spring Boot. The usage of the liveness probe alone forces us to take these delays into account and it can be tricky to set up parameters without compromising the fast response to the unhealthiness of applications. So if your container normally starts in more than initialDelaySeconds + failureThreshold × periodSeconds, you should specify a startup probe and use the same endpoint as the liveness probe.

4 Recipes

The following are some general rules for dealing with resource challenges.

4.1 Avoid Receiving Traffic during Shutdown

Kubernetes is a distributed system. From this follows that some actions are taken in parallel while deleting a Pod. So there is a possible time window where a Pod gets traffic while it is being deleted (or being in a shutdown process). A possible workaround is to add a preStop lifecycle hook to the Pod specification:

```
spec:
  containers:
  - name: main-container
    image: my-main-image
    lifecycle:
      preStop:
        exec:
          command: ["sh", "-c", "sleep 20"]
```

The selected sleep should be long enough for the app to execute all remaining tasks and responses. In the example case above, the Pod is immediately removed from Kubernetes Service, but does not receive a SIGTERM until 20 seconds have expired.

4.2 Scenario Spring Boot/JVM Applications

JVM applications are sometimes difficult to handle due to their high resource usage during initialization. It is also difficult to predict and limit memory requirements.

4.2.1 Why Do We Usually Have to Overprovision a JVM Container?

- JVM memory is divided into heap, non-heap, and other JVM internal memory (for garbage collector, IO buffers, metadata, native memory, Java threads, etc.).
- The Xmx option only limits heap. So the overall memory requirement must be determined by testing and is definitely higher than Xmx.
- The default JVM container detection sets Xmx to 25% of the resource limit setting for memory (depending on the JVM and for a large amount of memory).

For most cases, this default value for Xmx is too pessimistic and leads to costly overprovisioning. However, the total memory consumption of a JVM application can be estimated to be at least 25% higher than the Xmx setting (for a large amount of memory).

How do we get the JVM, resource limit, and resource request settings?

Step 0: Preliminary Work

First, we need to define what availability is in the context of our application: Which part of the application must respond in what time if we call it available, etc.

Define availability:

- Define availability for the whole platform (incl. SLOs[1] for traffic, latency, failure rate, MTTR).
- Break it down into individual service SLOs.
- Use the following tests to define resource limits and number of replicas.

The JVM has some embedded container awareness. Its default configuration is very pessimistic, so we should adopt it to get a realistic scenario.

[1] service level objectives, which are used to define the expectations for the behavior of the application

Customize JVM container awareness:

The JVM parameter *MinRAMPercentage* allows Xmx to be set for a small amount of memory (less than 256 MB and when MaxHeapSize / -Xmx is not set) to 50% as default. This is OK. For apps with the requirement for a large amount of memory (greater than 256 MB) set the *MaxRAMPercentage* parameter to 75%.

Both values are only a good first attempt and need to be validated by tests.

Step 1.1: Test without CPU Limits

To get an impression of how the application consumes memory and how its latencies behave we have to do some load tests.

Get application characteristics by testing...:

- ... with the best coverage possible
- ... with real requests as test data (e.g. httperf with replayed access log)
- ... to get maximum traffic possible while keeping compliance with the service SLOs

Step 1.2: Test with CPU Limits

The goal of this test step is to achieve the smallest CPU consumption possible (CPU limit should be as low as possible because we do not want to waste resources) and to get an idea of how many instances or replicas of an application are required.

1. Test/find the lowest CPU limit for **MTTR SLO compliance** (important, because JVM/Spring might need more resources for booting than for normal operation).
2. Test/find the lowest CPU limit when replaying the access log in real time for **compliance with latency and traffic SLOs**.

Some applications do not support more than one instance, due to issues with the database, synchronization, HTTP sessions, or whatever. So if your app does not support any replicas to achieve high availability, you are already done – just take the higher value of the two. However, such a hard limit of instances may require code and/or architecture changes to the application to get it more cloud-native.

If your app does support replicas, high availability comes with replicas and the MTTR requirement is already met. When it comes to the number of replicas to use we can consider the n+2 rule here, since Pods are logical hosts. So, if one Pod gets updated and one is allowed to crash, the remaining replicas must still be able to be compliant with the service's SLOs. Take the second value to estimate the number of Pod replicas required.

Step 2: Evaluate JVM Settings

Influence of CPU shares and memory limit on Garbage Collection:

Based on the available CPUs and memory, the JVM tries to select the best suitable Garbage Collector. Historically, the JVM knows two modes: A client mode for fat client applications and a server mode for long-running services.

Although not used in this way anymore, these modes influence the Garbage Collector selection. The JVM selects one of these modes automatically based on the CPU and memory.

Server mode is selected when the JVM has 1 CPU or more and 1792m or more memory. Everything below the client mode is selected. In client mode the JVM chooses the serial GC, in server mode the default G1GC (Garbage First Garbage Collector).

There are two ways to influence this decision manually as the CPU share is not equal in the container world just to the CPU time of one CPU. So even with a CPU share of 1024, the application could use several parallel threads.

- Option 1: Always run the application in server mode with *-XX:+AlwaysActAsServerClassMachine*. This indirectly also sets the default GC -> G1GC.
- Option 2: Set the Garbage Collector manually with *-XX:+UseG1GC*.

Step 3: Set Resource Requests and Limits

Finally, we can choose the required settings. In general, setting memory limits/requests is more critical than setting CPU limits/requests. The reason is that memory is a hard limit that can lead to "out of memory" exceptions and crashing of containers and/or nodes. Wrong CPU limits "only" lead to increasing latencies, which is in some way acceptable.

However, the chosen values are only a first attempt and must be further tested and monitored.

- Use the CPU limits from *1.2 Test with CPU limits*.
- Limit memory to the maximum RAM demand for our application container we got from testing. Plus a spare of about 10%.
- Use the same values for resource requests to achieve the **Guaranteed Quality of Service Class** (avoiding Pod eviction due to resource constraints).

4.3 Different Types of Workloads

In addition to the different priorities of each Pod, there are also different types of workloads to consider:

- Run to completion workloads like (backup) jobs, analytics, and calculations are automated through the resource types Jobs and CronJobs. Often the duration is not essential. So, set the CPU request to 1 so that the Pod does not get CPU shares when the total load on the node is high.
- Constantly running workloads until update/crash/downscaling etc. break down into:
  - Stateless workloads are automated through deployments. To benefit from features like rolling updates a deployment should even be used if the replica is 1.
  - Stateful workloads are automated by StatefulSets. They should be used by workloads that typically require ordered scaling and stable network identifiers (and persistence).

- Workloads with unique resource constraints (GPU, significantly more RAM or CPU than normal, etc.) should use a segmented cluster with specific worker nodes by configuring taints and node labels.

4.4 Planning Your Cluster

A great resource for planning the cluster is the Instance Calculator at https://learnk8s.io/kubernetes-instance-calculator. However, there are some additional ways of improving upscaling, deployment, and cluster size estimations.

4.4.1 Overprovisioning Recipes

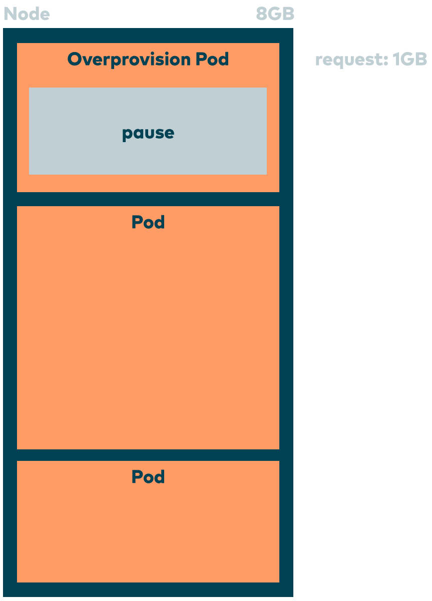
Overprovisioning at Cluster Level

It makes sense to overprovision clusters when...:

...we deploy certain workloads with frameworks that, for example, have higher resource requirements during boot phases than during normal operation. For Kubernetes this is achieved with setting ResourceLimits higher than the ResourceRequests. But this kind of overcommitment is dangerous. So additional overprovisioning can be risk mitigation.

...we are using HorizontalPodAutoscaling for some deployments. The reason for the issues here is how the cluster autoscaler usually works. It monitors the status of Pods and adds additional worker nodes when a Pod remains in pending state, because the cluster is running out of resources. In this case, it spawns a new node. Sounds good? Well, spawning nodes take minutes instead of the intended (milli-)seconds when spawning a Pod. This may be an issue in very dynamic environments. One solution could be to create a spare on each node and thus separate the creation of new nodes from the upscaling of Pods.

Overprovisioning clusters with overprovisioning Pods



One option to overprovision clusters is to add an overprovision Pod on every node. This Pod contains only a pause container, that simply calls pause on the Linux OS (pause causes the calling container process to sleep until a signal is delivered, so it does not consume any runtime resources).

The trick is to assign these overprovision Pods a lower priority (PriorityClass) than regular Pods so that they are evicted from their node as soon as resources become scarce (e.g. by scaling up a regular deployment). The pending state of the evicted overprovision Pod now initiates the scaling of worker nodes, although the regular Pod could be started immediately.

As shown in the figure, if we set the ResourceRequest for a memory parameter of the overprovision Pod to 1GB, we'll get a memory overprovisioning of ~12%. We can keep the dynamic nature of the cluster while scaling up and down by using a tool called cluster propositional autoscaler. It watches over the number of schedulable nodes and cores of the cluster and resizes the number of replicas for the overprovisioning Pods in this case. Details of its configuration can be found here: https://github.com/kubernetes/autoscaler/blob/master/cluster-autoscaler/FAQ.md#how-can-i-configure-overprovisioning-with-cluster-autoscaler

Overprovisioning at Pod Level

Typically, the application's use of resources such as memory or CPU changes depending on the type of functionality (e.g. different queries) and its load. So it's obvious that we have something like average resource utilization and peaks. So how do we treat these peaks? The solution is to define the average, typical resource usage and a kind of overprovisioning on Pod level as spare. What are the means we can use?

- Set the memory limit below request? Makes no sense, because containers get killed if they exceed their limit. So we do have a spare in this case, but it cannot be used by the application.
- (Mis-)use the Pod Overhead setting? Possible, but this is a Pod-level attribute, not container level. On the other hand, if a couple of similar Pods consist of a main container and some sidecars it is possible to add a RuntimeClass instance for these Pods with a (Pod)Overhead, hence overprovisioning, for the whole Pod while keeping guaranteed QoS (same value for limits and requests).
- Adopt framework settings (e.g. JVM: Refer to Why Do We Usually Have to Overprovision a JVM Container?).
- Adopt tool settings to keep memory or CPU below ResourceLimits (e.g. MongoDB, Prometheus).

4.5 Deployment Recipes

There are several deployment strategies, some with downtime like Recreate, some without like Rolling Update or Canary. Some releases are even shifted to testing in production by using feature toggles.

But when we look at production, there is a certain hierarchy of how we want to release new versions.

> First of all, Recreate is the cheapest and simplest strategy. If we use Recreate, we have time to remove the outdated deployment, do cleanup, update the DB schema, ETL and so on.

So it's certainly a good idea that your application architecture and availability SLOs allow for as many Recreate deployment strategies for their application Pods as possible. We can accomplish this by defining a practical/pragmatic availability (which we break down to individual service SLOs) and by applying the CQRS[2] and/or the Microservices pattern.

However, if downtime is an issue, and for some services it will be unavoidable, we need to deploy differently. Basically we have two to choose from: Rolling Update or Canary release. In the context of Kubernetes, Rolling Update is a built-in feature and refers to updating the container image version of a deployment or a StatefulSet. So determine how many Pod instances are required to meet the specified service SLOs and set the Pod Disruption Budget accordingly.

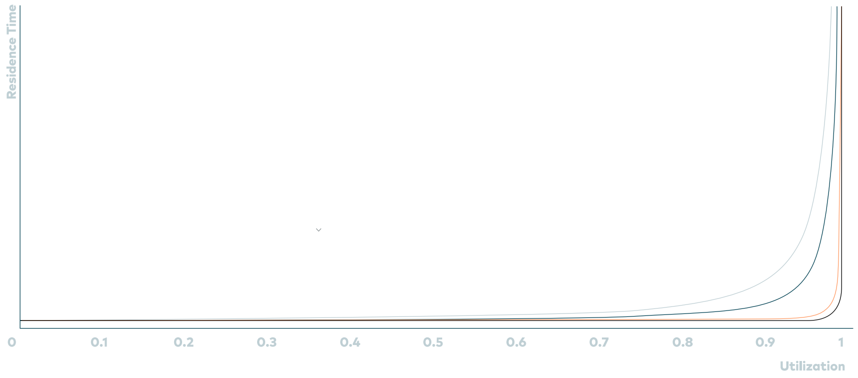
During such a rollout, however, the amount of traffic such a new version receives is tied to the number of instances. If this is too coarse-granular, other tools must be used. That's why Service Meshes introduced a virtual service concept to allow a routing-based Canary release. Virtual Services move user traffic from one deployment in version V1 to another in version V2, a kind of fine-grained Blue-Green release while being independent of the Kubernetes Deployment itself.

4.6 Cluster Sizing Recipes

The size of a cluster is an important cost factor in production (besides storage, managed services, and traffic). So the size of a cluster has to be weighted between overprovisioning (sufficient spare space for the applications to mitigate high load) and cost. There are some basic rules of thumb to estimate cluster size and required size increase on higher load.

[2] Intro to CQRS by Martin Fowler: https://martinfowler.com/bliki/CQRS.html

n+2 Rule



This figure shows response time curves of systems with 1, 2, 4, 32, and 64 servers. The so-called knee (sometimes explained by the queuing theory) is known to be at ~80% utilization. While this is only valid for low resources (which is typically the case for nodes used as cluster nodes), you shouldn't run your servers past that point or performance will begin to degrade.

The n+2 rule says that if we generalize this 80% target and one node is unavailable because it gets maintenance updates, and one node is allowed to fail then the remaining nodes should only have a resource utilization of maximum 80%.

For example, this results in a normal operation of a maximum ~50% resource usage for five nodes (3*80%/5). So for a machine type with two vCPUs a sum of Kubernetes ResourceRequest/ResourceLimit of five vCPUs are usable. However, this is just a first estimation.

The Square Root Staffing Law

The square root staffing law is derived from queuing theory and is useful for getting an estimate of the capacity you need to serve an increased volume of traffic. It predicts that the spare capacity needed for a specified QoS grows in proportion to the square root of the traffic increase.

For the scenario taken from n+2 Rule, we have five nodes with an overprovision of 50%. In other words, 2.5 fully loaded nodes and 2.5 spare nodes. If for example we assume a 200% increase in traffic, we get an estimate of 2.5 * 200% + 2.5 * SQR(200%) = 5 fully utilized nodes with ~4 nodes as reserve = 9 nodes.

4.7 Cluster Segmentation

The purpose of Kubernetes is resource abstraction. This allows us to focus on applications rather than hardware such as worker nodes or node types. The more homogeneous the cluster, the better we can achieve this goal. However, sometimes the traditional VM zone pattern plays a role, where we separate frontend VMs from backend VMs via different networks and firewalls in between. But the segmentation of worker nodes should not be applied to a node abstraction pattern like Kubernetes. The right equivalent concept is Network Policies, which are able to implement a logical zone/project/team/role concept, including firewalls, around Pods. It is part of the necessary persuasion work to ensure that Network Policies are accepted by security departments; after all, traditional VMs also run on the same hardware. The discussion of container security versus VM security might be an important discussion, though.

A more reasonable justification for segmenting a cluster is different hardware requirements by very special workloads, such as machine learning or CI/CD. ML might require hardware with GPUs; CI might require much more continuous memory than the rest of the application. Kubernetes has two means for that: taints and tolerations and node labels. So, use node labels to flag the existence of specific hardware and pin the respective workloads to these nodes via labels (attribute **nodeSelector**). If it makes sense to reserve these nodes exclusively for special workloads, use a taint to allow the node to reject other Pods that do not tolerate this taint.

4.8 Availability Rules

In general, high availability is generated by using the concept of node or Pod replicas. However, availability must be addressed on several levels.

First, the application or tool used must be able to handle replicas (and a round-robin load balancer up front). In particular, stateful applications or applications that use HTTP sessions for example are often initially unable to do so.

Second, using cloud vendor zones can increase availability because it is unlikely that more than one zone goes down simultaneously. However, this has implications for stateful applications with volumes. Typically, volumes cannot be moved between zones. This results in the need for more than one node per zone (at least two nodes) to create high availability in the zone itself, and a node autoscaler that takes this volume limitation into account.

Rules:

- If the Recreate deployment strategy is possible (this is the simplest strategy from a production point of view), there is no need for replicas. At least when it comes to availability. For performance reasons, this may still be appropriate.
- The use of zones and replicas per zone is mostly recommended. Most tools like Kubernetes and node-autoscaler support this concept. On the other hand, if the availability of one zone is sufficient it may be simpler to use only a single zone while keeping a spare cluster/tool/application on another zone as hot standby. In this case, the simplicity gained, required availability, and switchover/backup/recovery times must be weighed against each other.
- If you use tools (like Kafka or databases), ensure that the configuration of its replicas really increases availability. Otherwise, you end up with dependent replicas that significantly reduce availability instead of increasing it.
- Use Pod anti-affinities (that's default) to ensure that replicas are running on different worker nodes. Otherwise, it is still high availability on Pod level (e.g. during Rolling Update), but does not help if the entire node dies.

4.9 Node Type Rules

If you are creating a Kubernetes cluster, you must specify the size of the worker nodes. Of course, the many different types of workloads play an important role. It is therefore not easy to come up with generally applicable rules.

However, the following statements can be made:

Bigger and therefore fewer nodes:

- Less inter-node traffic
- Less control plane overhead, but more overhead on node due to Kubelets container probes, cAdvisor stats collection, etc. (more Pods per node)
- Typically lower costs per workload, but bigger increments (worker node up-scaling)
- Allows running resource-intensive workloads
- Larger impact overall on availability if a node dies
- Typically higher IOPS[3] per node

Smaller and therefore more nodes:

- Smaller increments but more inter-node traffic, etc.
- Lower Pod count limits due to instance type restrictions (e.g. **AWS?** a t2.micro has a max. of four Pods)
- Typically lower IOPS per node

Some general rules:

- It is always better not to segment the cluster.
- For a few workloads with higher resource usage (e.g. CI pipelines, machine learning) without direct impact on application availability, consider cluster segmentation with node labels and taints on one or two bigger node types.
- For bigger workloads (> 500MB) choose bigger machine types (> 8GB) and/or machine types with higher RAM vs CPU weighting.
- For smaller workloads (< 100MB) choose smaller machine types (< 8GB) and/or machine types with higher CPU vs RAM weighting.
- For workloads with heavy file-IO and/or more CPU load choose bigger machine types.
- It rarely makes sense to use bare metal machines with for example 128GB RAM and 64 cores and Pods using 10MB RAM each.

[3] input/output operations Per second

4.10 Pod Priorities

The Pod's Quality of Service attribute is derived from its configuration of resource requests vs. resource limits. Kubernetes decides which Pod can be evicted once the node gets into a low memory condition (by other components on the node or by other Burstable Pods using more memory defines in its resource requests). The order of eviction is: Best Effort Pods are evicted first, followed by Burstable Pods. Guaranteed Pods will never be evicted because of another Pod's resource consumption (in contrast to node pressure eviction, where this can still happen).

However, if Burstable Pods fill up a node, they will not be evicted even by applying Guaranteed Pods. If there is no space on the cluster, they remain in pending state until enough space is available again.

> QoS is therefore not a suitable means of distinguishing between important and less important application components.

If less important Pods are to be displaced by more important Pods, the so-called Pod Priorities must be used. This guarantees that a higher-priority Pod will be deployed (e.g. in case of automated upscaling) at the expense of lower priorities (or no priority) if necessary.

Rules:

- Use as many Guaranteed Pods as possible (resource requests equals resource limits).
- Keep in mind that Burstable/Best Effort Pods are eligible for eviction due to resource pressure on the node.
- Distinguish between important and less important Pods based on availability requirements and set Pod Priorities accordingly.

About the Authors

Christopher Schmidt

@fakod

Christopher is a senior consultant at innoQ Schweiz GmbH. He has been at home in software development for more than 20 years. During this time he has successfully brought numerous software and modernization projects into production in various roles. Christopher's focus is on current front / back-end technologies and highly scalable architectures. Kubernetes is his passion.

Christine Koppelt

@ckoppelt

Christine Koppelt works as a Senior Consultant at INNOQ Germany. Her focus is on the implementation and modernization of digitization projects for medium-sized companies. She is particularly interested in software architecture, infrastructure and data engineering.